HoGent

BEDRIJF
EN
ORGANISATIE

Builder

Make and return one object various ways.

Definitie

- Gebruik het Builder pattern om de constructie van een product af te schermen en zorg dat je het in stappen kan construeren

Ik wil een sandwich

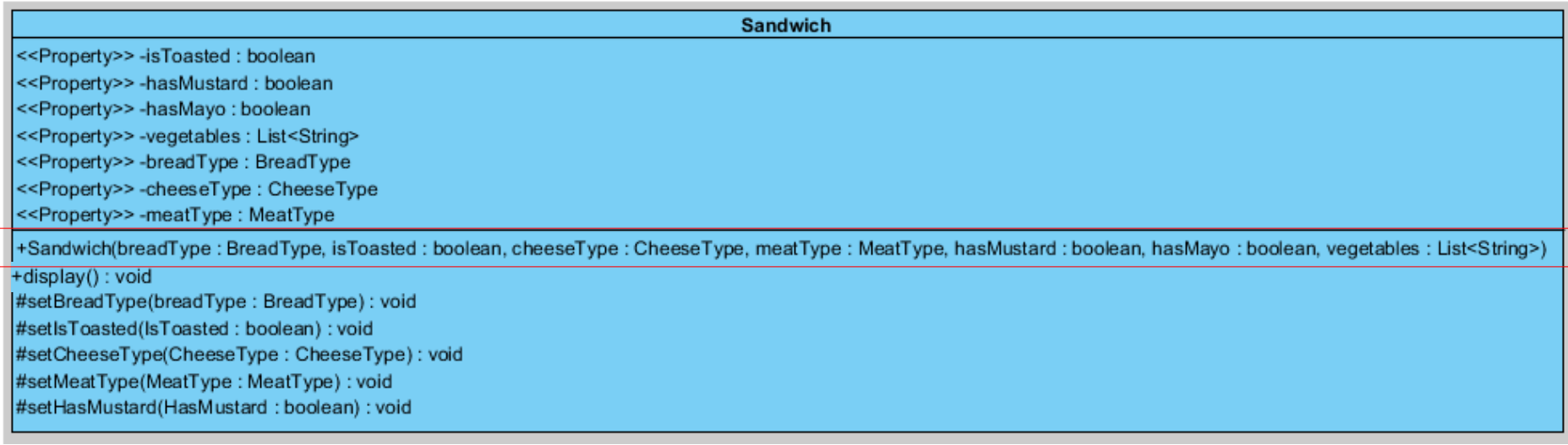
- Hey?
  - Hey?
- Ik wil een sandwich!
  - Ok, welk brood wens je? Wit of bruin?
- Wit
  - Ok, welke grootte?
- Een medium
  - Ok, welk soort vlees en kaas?
- Kalkoen en jonge kaas.
  - Mag het worden opgewarmd?
- Ja
  - Welke groentjes?
- …

Ik wil een sandwich – een andere manier

- Hey?
  - Hey?
- Ik wil een sandwich, de super turkey pickler!
  - Ok
- De kok volgt zijn ingredientenlijst om de sandwich te maken

Het voorbeeld

- Klasse Sandwich



Het voorbeeld

- Klasse Sandwich

```
public class Sandwich {

  private BreadType breadType;   private boolean isToasted;
  private CheeseType cheeseType;   private MeatType meatType;
  private boolean hasMustard;   private boolean hasMayo;
  private List<String> vegetables;

  public Sandwich(BreadType breadType, boolean isToasted, CheeseType cheeseType,
      MeatType meatType,  boolean hasMustard, boolean hasMayo,
      List<String> vegetables) {
    setBreadType(breadType);
    setIsToasted(isToasted);
    setCheeseType(cheeseType);
    setMeatType(meatType);
    setHasMustard(hasMustard);
    setHasMayo(hasMayo);
    setVegetables(vegetables);
  }
```

```
public BreadType getBreadType() {
  return breadType;
}


protected void setBreadType(BreadType breadType) {
  this.breadType = breadType;
}
```

Het voorbeeld

- Een sandwich maken

```
public static void main(String[] args) {
    List<String> vegetables = new ArrayList<>();
    vegetables.add("Tomato");
    vegetables.add("Salad");
    Sandwich sandwich
            = new Sandwich(BreadType.Wheat, false, CheeseType.American,
                    MeatType.Turkey, false, false, vegetables);
    sandwich.display();
}
```

Probleem 1 : te veel parameters

- Te veel parameters
- Vaak ook meerdere constructors

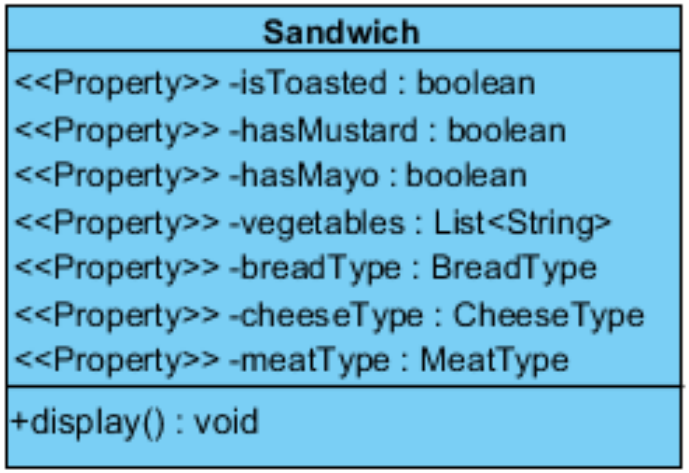
- Oplossing 1 : De waarden laten instellen via setters
  - Klasse Sandwich : enkel de default constructor voorzien en alle setters public maken

- Voordelen
  - Constructor met vele parameters weg

- Maar nu creëren we een nieuw probleem

Oplossing 1

- Klasse Sandwich




```
public class Sandwich {

   private BreadType breadType;
   private boolean isToasted;
   private CheeseType cheeseType;
   private MeatType meatType;
   private boolean hasMustard;
   private boolean hasMayo;
   private List<String> vegetables;

   public BreadType getBreadType() {
      return breadType;
   }

   public void setBreadType(BreadType breadType) {
      this.breadType = breadType;
   }
```

Oplossing 1

- Een sandwich maken

```
public static void main(String[] args) {
    List<String> vegetables = new ArrayList<>();
    vegetables.add("Tomato");
    vegetables.add("Salad");
    Sandwich sandwich = new Sandwich();
    sandwich.setBreadType(BreadType.Wheat);
    sandwich.setCheeseType(CheeseType.American);
    sandwich.setMeatType(MeatType.Turkey);
    sandwich.setHasMayo(false);
    sandwich.setIsToasted(true);
    sandwich.setHasMustard(true);
    sandwich.setVegetables(vegetables);
    sandwich.display();
}
```

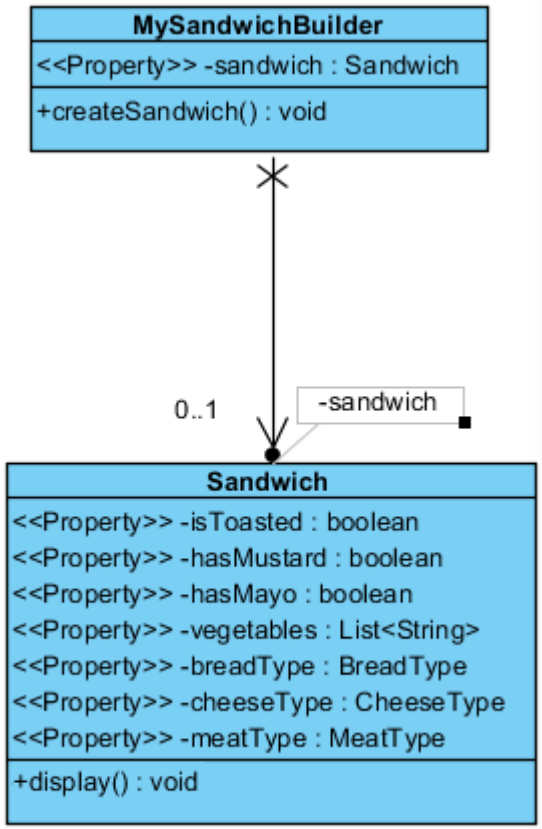
Probleem 2 : volgorde afhankelijk

- Wat als we een aantal properties niet (vergeten) instellen?
- We moeten de stappen kennen om een sandwich te creëren. (Wat als we de properties in een verkeerde volgorde een waarde toekennen?)

- Oplossing : we maken onze eigen sandwich builder
- Voordeel : het maken van een sandwich bestaat uit een aantal stappen die in een bepaalde volgorde moeten worden uitgevoerd. Validatie kan worden toegevoegd

- Nadeel : we bouwen 1 specifieke sandwich

Oplossing 2




```java
public class MySandwichBuilder {
  private Sandwich sandwich;
  public Sandwich getSandwich() {
    return sandwich;
  }
  public void createSandwich() {
    List<String> vegetables = new ArrayList<>();
    vegetables.add("Tomato");
    vegetables.add("Salad");
    sandwich = new Sandwich();
    sandwich.setBreadType(BreadType.Wheat);
    sandwich.setMeatType(MeatType.Turkey);
    sandwich.setCheeseType(CheeseType.American);
    sandwich.setHasMayo(false);
    sandwich.setIsToasted(true);
    sandwich.setHasMustard(true);
    sandwich.setVegetables(vegetables);
  }
}
```

Bevat een sandwich

Bevat een methode om de sandwich op te vragen

Bevat methode die sandwich creëert

Oplossing 2

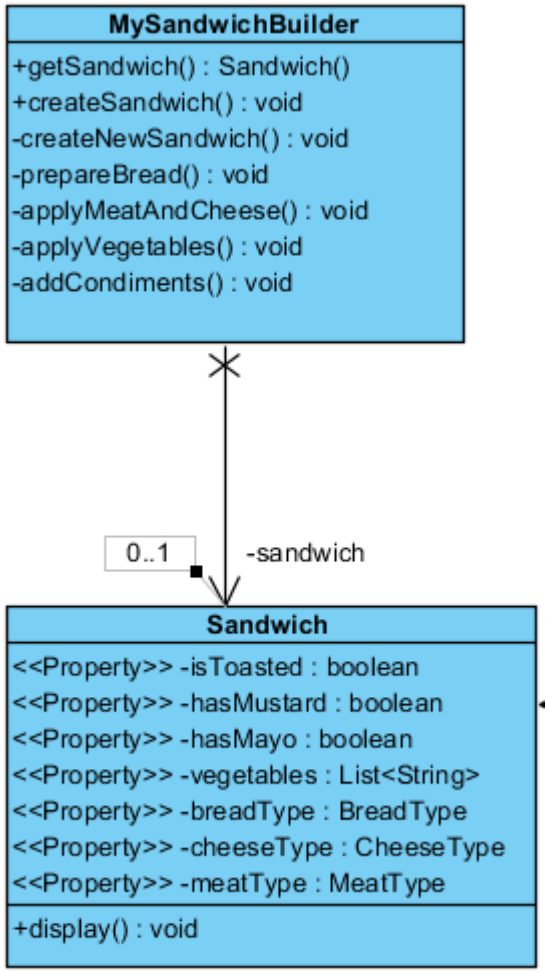
- Een sandwich maken

```
public static void main(String[] args) {
    MySandwichBuilder builder = new MySandwichBuilder();
    builder.createSandwich();
    Sandwich sandwich = builder.getSandwich();
    sandwich.display();
}
```

Oplossing 2 :

- Proces toevoegen
  - Het bouwen van een sandwich bestaat uit een aantal stappen.
  - We gaan de code in die zin aanpassen

Oplossing 2 : proces toevoegen

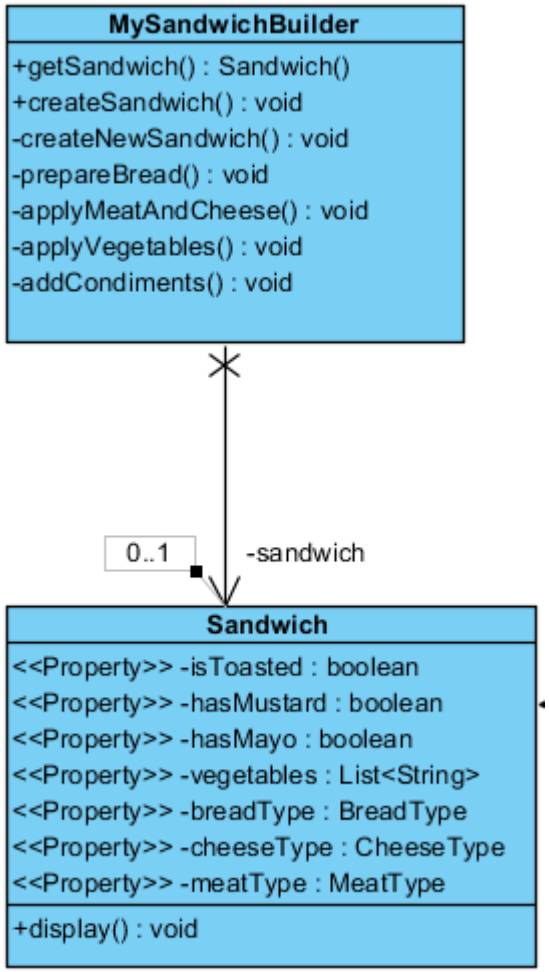



```
public class MySandwichBuilder {
  private Sandwich sandwich;
  public Sandwich getSandwich() {
    return sandwich;
  }
  public void createSandwich() {
    createNewSandwich();
    prepareBread();
    applyMeatAndCheese();
    applyVegetables();
    addCondiments();
  }
  private void createNewSandwich()
  {
    sandwich = new Sandwich();
  }
```

Het bouwen van de sandwich in opeenvolgende stapjes

Elke stap is een aparte methode

Oplossing 2 : proces toevoegen




```
private void prepareBread() {
    sandwich.setBreadType(BreadType.Wheat);
}
private void applyMeatAndCheese() {
    sandwich.setCheeseType(CheeseType.American);
    sandwich.setMeatType(MeatType.Turkey);
}
private void applyVegetables() {
    List<String> vegetables = new ArrayList<>();
    vegetables.add("Tomato");
    vegetables.add("Lettuce");
    sandwich.setVegetables(vegetables);
}
private void addCondiments() {
    sandwich.setHasMayo(false);
    sandwich.setIsToasted(true);
    sandwich.setHasMustard(true);
}
```

Probleem 3 : verschillende constructies

- We bouwen maar 1 specifieke sandwich. Wat als we nu ook een ClubSandwich willen bouwen?

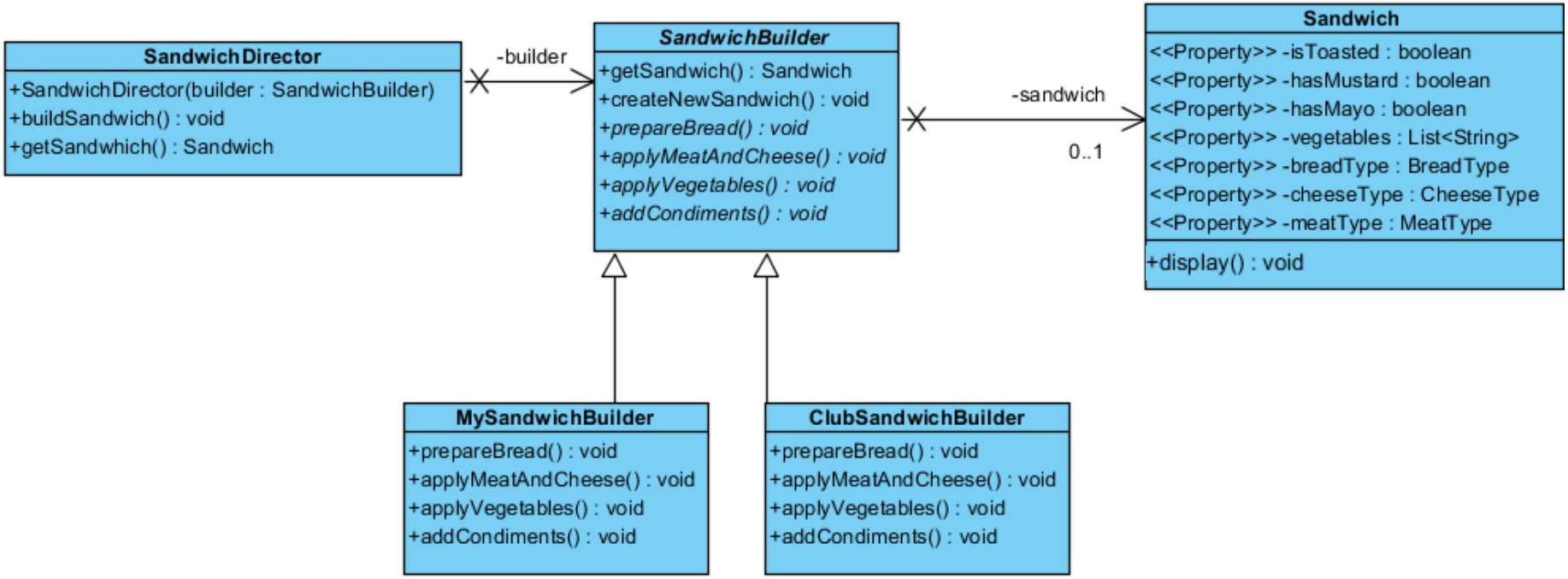
Je hebt een flexibel ontwerp nodig

- De sandwich kan voor elke klant verschillen. Sommigen wensen geen groenten, anderen geen mayo,…
- Je hebt een flexibele datastructuur nodig die een sandwich voor elke klant in al zijn variaties kan representeren. Bovendien moet je een aantal stappen volgen om een sandwich te maken.
- Hoe kan je een manier vinden om een complexe structuur te maken zonder deze te vermengen met de stappen die nodig zijn om deze structuur te maken?

Je hebt een flexibel ontwerp nodig

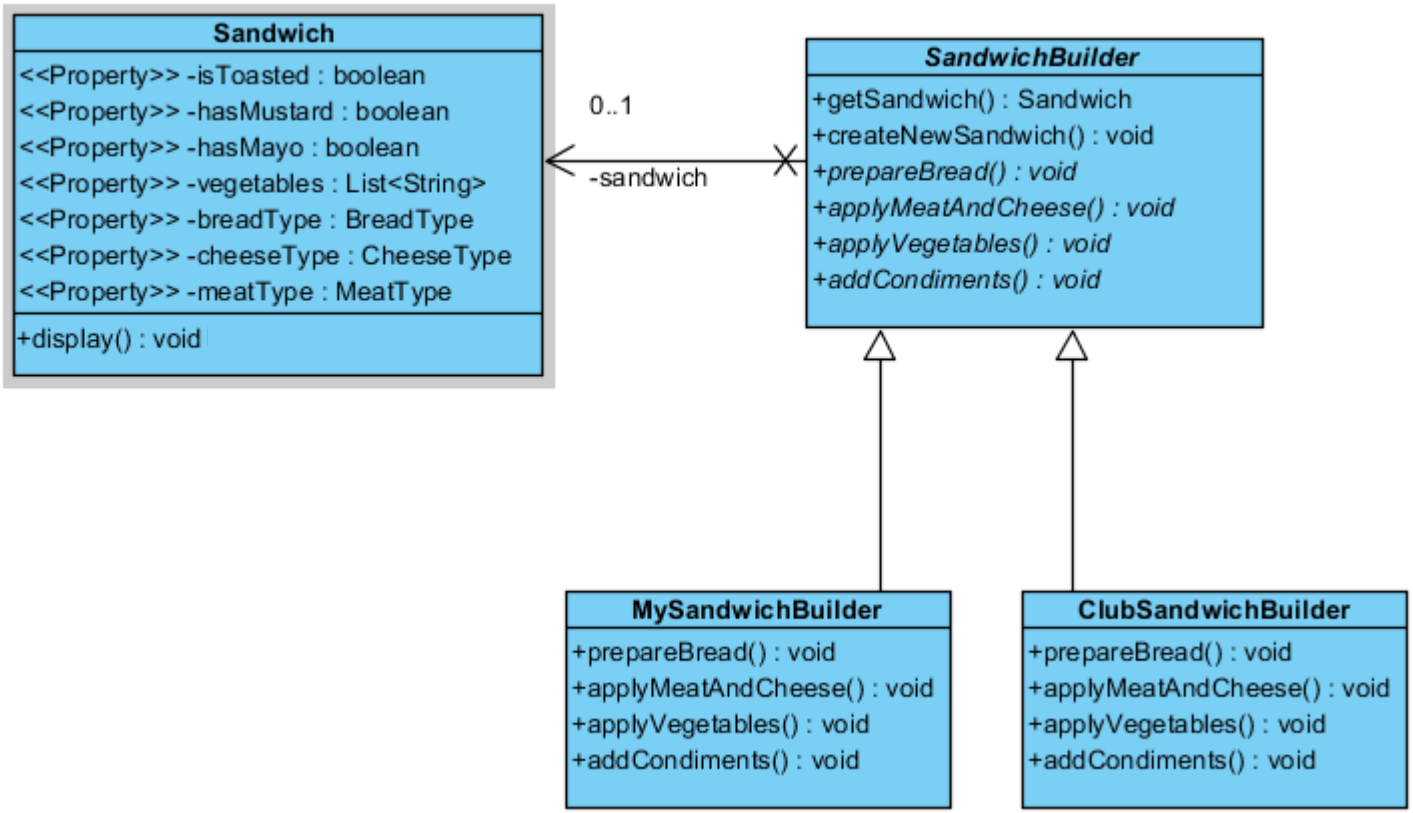
- Herinner je je de Iterator?
  - De iterator werd afgeschermd in een aparte object en verborg de interne representatie van een collectie voor de client
- Wat als we hetzelfde idee hier eens zouden gebruiken?
  - we schermen het maken van een sandwich af in een object (laten we het de **builder** noemen). Deze bevat alle methodes die nodig zijn voor het bouwen van de onderdelen (of de verschillende stappen) van een sandwich
  - We maken een **director**, die de stappen voor het bouwen van een sandwich kent en die aan de builder vraagt de structuur van de sandwich te maken

Je hebt een flexibel ontwerp nodig



De builder

- We bouwen een interface voor het bouwen van een sandwich, met alle methodes voor het bouwen van de onderdelen



De builder

- De abstracte klasse

```
public abstract class SandwichBuilder {

   private Sandwich sandwich;
   public Sandwich getSandwich() {
      return sandwich;
   }
   public void  createNewSandwich()  {
       sandwich = new Sandwich();
   }
   public abstract void prepareBread() ;
   public abstract  void applyMeatAndCheese();
   public abstract  void applyVegetables() ;
   public abstract  void addCondiments() ;
}
```

De builder klassen

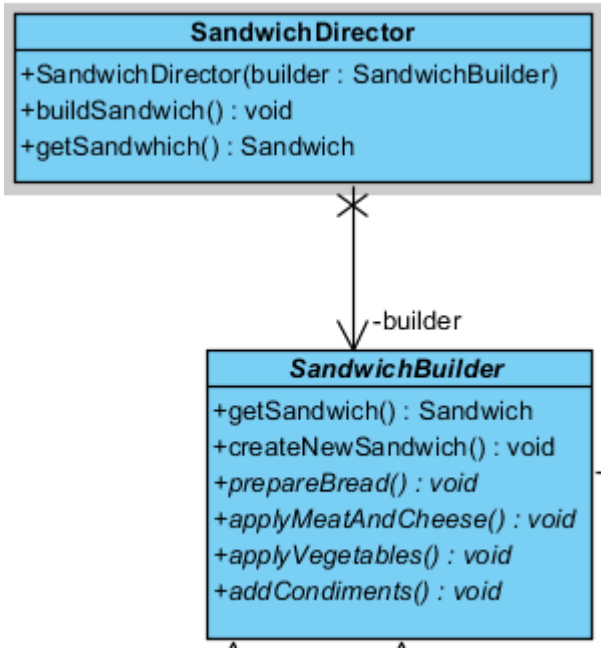
- De concrete klassen

```
public class MySandwichBuilder extends SandwichBuilder {
   public void prepareBread() {
      Sandwich sandwich = getSandwich();
      sandwich.setBreadType(BreadType.Wheat);
   }
   public void applyMeatAndChee
      Sandwich sandwich = getSand
      sandwich.setCheeseType(Che
      sandwich.setMeatType(Meat
   }
   public void applyVegetables() {

   public void addCondiments() {.
}
```

```
public class ClubSandwichBuilder extends SandwichBuilder {
    public void prepareBread() {
     Sandwich sandwich = getSandwich();
     sandwich.setBreadType(BreadType.White);
   }
   public void applyMeatAndCheese() {
     Sandwich sandwich = getSandwich();
     sandwich.setCheeseType(CheeseType.Swiss);
     sandwich.setMeatType(MeatType.Ham);
   }
   public void applyVegetables() {...8 lines }

   public void addCondiments() {...6 lines }
}
```

De Director

- Kent het proces om een sandwich te maken, ongeacht het type sandwich. Dit laat hij over aan de builder

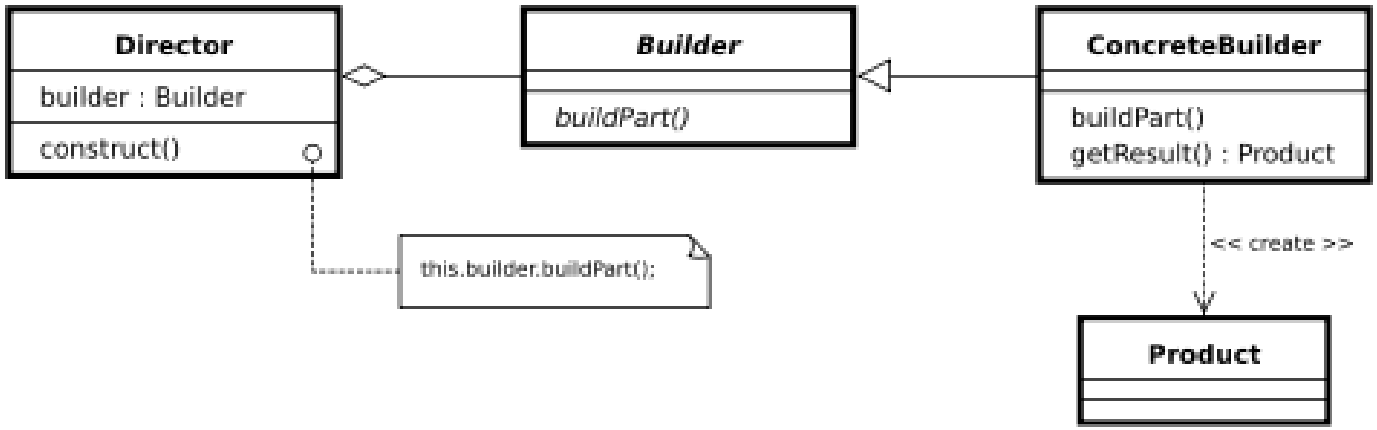



```
public class SandwichDirector {
  private  SandwichBuilder builder;
    public SandwichDirector(SandwichBuilder builder)  {
      this.builder = builder;
    }
     //uses builder to create sandwich
    public void buildSandwich()     {
      builder.createNewSandwich();
      builder.prepareBread();
      builder.applyMeatAndCheese();
      builder.applyVegetables();
      builder.addCondiments();
    }
    //get sandwich back out
    public Sandwich getSandwhich()     {
      return builder.getSandwich();
    }
}
```

Let's make some sandwiches ☺

```
public static void main(String[] args) {
    SandwichDirector director  = new SandwichDirector(new MySandwichBuilder());
    director.buildSandwich();
    Sandwich sandwich = director.getSandwhich();
    sandwich.display();
}
```

Het Builder pattern



- De **Builder** klasse specifieert een abstracte interface voor de creatie van de onderdelen van het Product object.
- De **ConcreteBuilder** bouwt de onderdelen van het complexe object en gooit deze samen door implementatie van de Builder interface. Het houdt een representatie van het object bij en biedt een interface voor het opvragen van het product.
- De **Director** class bouwt het complexe object gebruik makend van de interface van de Builder
- De **Product** stelt het complexe object voor dat gebouwd wordt.

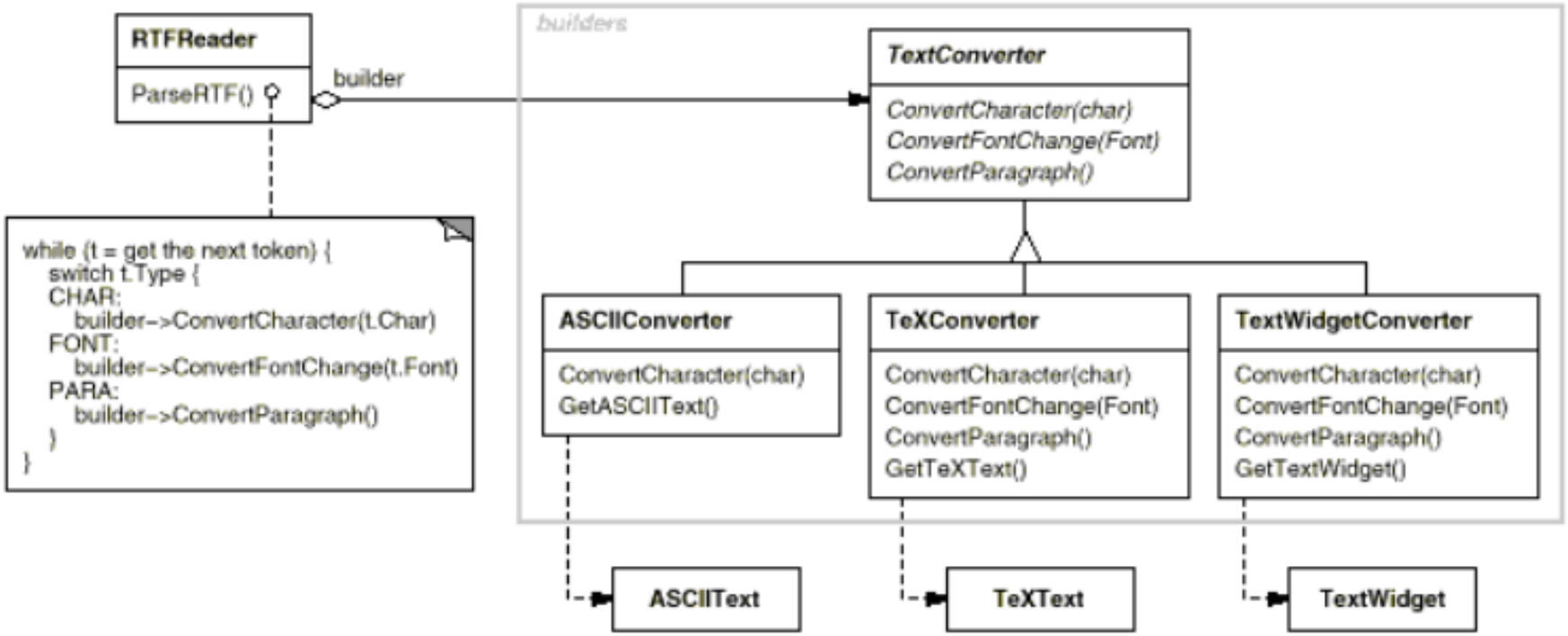
Voordelen van builder

- Schermt de manier waarop een complex object gebouwd wordt af
- Geeft de mogelijkheid om objecten in meerdere stappen en wisselende processen te maken (in tegenstelling tot een éénstapsfactory)
- Verbergt de interne representatie van het product voor de client
- Productimplementaties kunnen steeds wisselen, omdat een client alleen een abstracte interface ziet

Gebruik en nadelen Builder

- Wordt vaak gebruikt voor samengestelde objecten
- Het maken van een object vereist meer domeinkennis van de client (tenzij je een Director klasse kan voorzien) dan wanneer je een Factory gebruikt.

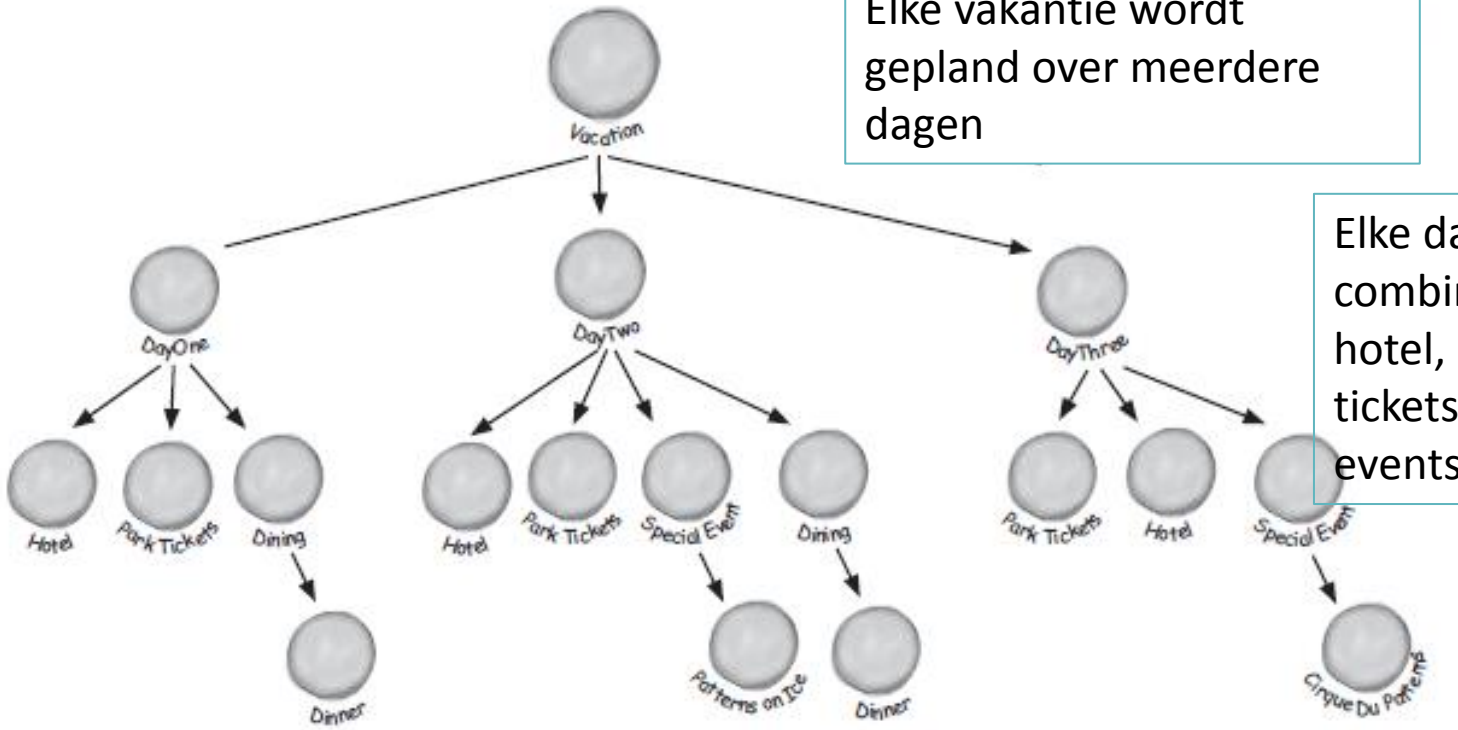
Extra voorbeeld



Erich, Gamma; Helm, R. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*., p110

Extra voorbeeld

- Bouw een vakantieplanner voor Patternsland.
  - Elke gast kan een hotel kiezen en verschillende toegangsticketten, een restaurant reservatie maken en zelfs speciale events boeken



Elke vakantie wordt gepland over meerdere dagen

Elke dag kan een combinatie van hotel, restaurant, tickets en speciale events bevatten

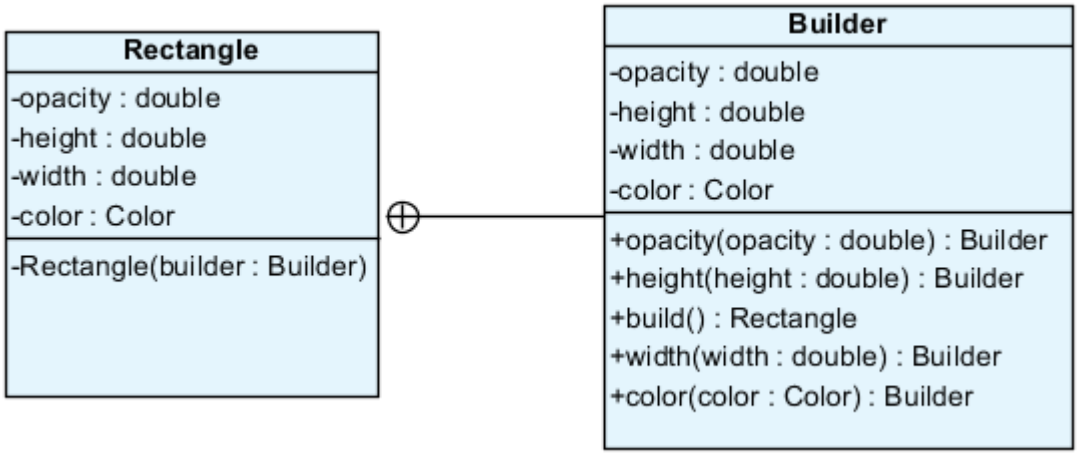
Een variant

- Voor de constructie van klasse
  - met veel opties
  - Met veel argumenten bij de constructie
  - Met mogelijks ook al veel default waarden
- Maak gebruik van een builder, als interne klasse

```
public static void main(String[] args) {
   Rectangle r =
         new Rectangle.Builder().height(250)
               .width(300).opacity(0.5).color(Color.PINK).build();
}
```

Een variant

- Maak gebruik van een builder, als interne klasse



Een variant

- Klasse Rechthoek

```
public class Rectangle {
    private final double opacity;
    private final double height;
    private final double width;
    // ...
    private final Color color;
    public static class Builder {...33 lines }
    private Rectangle(Builder builder) {
        this.opacity = builder.opacity;
        this.height = builder.height;
        this.width = builder.width;
        this.color = builder.color;
        //...
        //Hier validatie code...
    }
}
```

Een variant

- Klasse Rechthoek

```
public static class Builder {
   private double opacity;
   private double height;
   private double width;
   private Color color;
   // ...
   public Builder opacity(double opacity) {
      this.opacity = opacity;
      return this;
   }
   public Builder height(double height) {
      this.height = height;
      return this;
   }
   public Builder width(double width) {
     this.width = width;
     return this;
   }
   public Builder color(Color color) {
      this.color = color;
      return this;
   }
  // ...

   public Rectangle build() {
      return new Rectangle(this);
   }
}
```